DefiningUser

Postgres has a concept of "user-defined types". Internally, Postgres regards a user-defined type as a blob of memory that your functions know everything about, in terms of processing. Postgres will store and retrieve the data from disk and use your functions to input, process, and output the data.

1. User

A completely defined type includes the following:

- o Input and output functions for the type.

These functions determine how the type appears in strings (for user input and output) and how the type is organized in memory. For more details about user-defined functions, see the tutorial "Defining a User Defined Function".

- o Operator functions for the type.

These functions define the meanings of "equal", "less-than", "greater-than", etc for your type.

1.1. Creating

In this discussion, we will be defining a "circle" type. Before we do anything, we have to decide on what a circle looks like, both in string format and internally in memory. Circles have a center and a radius, so a reasonable way to define a circle is:

(center_x, center_y, radius)

in whatever Cartesian units we desire. Now we essentially have defined what the input to our circle input function looks like, and what the output from the circle output function should look like. Now we have to come up with an internal representation for a circle in memory. Note that as this memory will be written to disk and read from disk, our internal representation must be both COMPLETE and CONTIGUOUS - that is, we can have no pointers within our structure. The following declaration for a circle is legal and reasonable given our format described above:

```
typedef struct
{
   double x,y;
}
POINT;

typedef struct
{
   POINT center;
   double r;
}
CIRCLE;
```

Just for the sake of argument, while the following declaration may seem reasonable, it is NOT, as Postgres will not be able to find the right "center" (because it is a volatile pointer) once it has written this information to disk:

```
typedef struct
{
   POINT *center;
   double r;
}
CIRCLE;
```

where POINT is as in the first example.

Suppose in defining our type "circle", we have a file called "circle.c" and it is in /usr/postgres/tutorial/circle.c. It has been compiled, and the compiled object is in /usr/postgres/tutorial/circle.o. Note that as mentioned above, all functions related to our "circle" type have to be in one object (".o") file. The source code for circle.c is at the end of this file.

For the sake of argument, suppose we are on a DECstation, where sizeof(double) is 8 bytes (this will be important later).

In "circle.c", there are three functions:

```
circle_in, which is the input function for circles. It takes a string as
        input and outputs a pointer to type CIRCLE. It must know how to
        parse the string itself.

circle_out, which is the output function for circles. It is passed a pointer
        to type CIRCLE as input and returns a string. This string must be
        suitable for input to circle_in. That is, the following piece
        of code should work:

        CIRCLE *circle, *circle_in();
        char *circle_out();

        circle = circle_in(circle_out(circle_in("(2.0, 3.0, 5.0)")));

equal_area_circle, which is the equality function for circles. For the
            purposes of this discussion, circles are equal if their
            areas are equal.
```

Now that we have written these functions and compiled them, we have to let Postgres know that they exist. We will do this first by running the following queries to define the input and output functions:

```
define C function circle_in (file = "/usr/postgres/tutorial/circle.o",
   returntype = circle) arg is (char16)

define C function circle_out (file = "/usr/postgres/tutorial/circle.o",
   returntype = char16) arg is (circle)
```

Now that we have done this, we can define the type. Note that the in and out functions have to be defined BEFORE we define the type. Defining the type first will not work. Postgres will notify you that return type circle is not defined yet, but everything is OK. The query to define the "circle" type is:

```
define type circle (internallength = 24, input = circle_in, output = circle_out)
```

where internallength is the size of the circle structure itself in bytes. For circles, the arguments are three doubles, thus

```
sizeof(circle) = 3 * sizeof(double) = 3 * 8 = 24.
```

When defining your own types, you should write a test program that does a

```
printf("size is %d0, sizeof(structure));
```

if you encounter strange errors in the use of your type, to determine that internallength is correct.

Now we have finished defining the "circle" type. Now we can create relations with circles in them, append records to them with circles defined, and retrieve the values of the entire list of records. But we can do nothing else until we have some circle operators. We do this through a concept called "operator overloading", and in this case we will set the Postgres equality operator "=" to work for circles. First we have to tell Postgres that our circle equality function exists with the following query:

```
define C function equal_area_circle (file = "/usr/postgres/tutorial/circle.o",
   returntype = bool) arg is (circle, circle)
```

Now that we have done this, we now can bind this function to the equality symbol with the following query:

```
define operator = (arg1 = circle, arg2 = circle, procedure = equal_area_circle)
```

Now we are ready to run some queries with our new type:

```
create tutorial (a = circle)

append tutorial (a = "(1.0, 1.0, 10.0)"::circle)

append tutorial (a = "(2.0, 2.0, 5.0)"::circle)

append tutorial (a = "(0.0, 1.8, 10.0)"::circle)

retrieve (tutorial.all) where tutorial.a = "(0.0, 0.0, 10.0)"::circle

--------------------
|        a         |
--------------------
| (1.0, 1.0, 10.0) |
--------------------
| (0.0, 1.8, 10.0) |
--------------------
```

Recall that we defined circles as being equal if their areas are equal.

Other operators (less than, greater than, etc) can be defined in a similar way. Note that the "=" symbol will still work for other types - it has merely had a new type added to the list of types it works on.

2. A

2.1. Use

When allocating space to be returned to Postgres, be sure to use the Postgres memory manager "palloc()". Postgres expects that all memory allocated during a transaction is allocated with "palloc()" and will give strange errors if "malloc()" or some other allocator is used directly. The interface to "palloc" are the same as to malloc, ie

```
char *palloc(size)

unsigned long size;
```

Along with "palloc", there is a call "pfree" which frees memory allocated using "palloc". Its interface is the same as "free()", ie

```
void pfree(address)

char *address;
```

where address was allocated with palloc.

The following is the source code for circle.c.

```
#include <stdio.h>

typedef struct
{
   double x, y;
}
```

```
POINT;


typedef struct
{
    POINT    center;
    double   radius;
}
CIRCLE;  /* This has to be a contiguous, wholly defined structure */

CIRCLE  *circle_in();
char    *circle_out();
char    circle_equal();

#define LDELIM    '('
#define RDELIM    ')'
#define NARGS      3

CIRCLE *
circle_in(str)
char    *str;
{
    double    atof(),    tmp;
    char      *strcpy(), *p, *coord[NARGS], buf2[1000];
    int    i;
    CIRCLE    *result;

    if (str == NULL)
        return(NULL);
    for (i = 0, p = str; *p && i < NARGS && *p != RDELIM; p++)
        if (*p == ',' || (*p == LDELIM && !i))
            coord[i++] = p + 1;
    if (i < NARGS - 1)
        return(NULL);

/*
 * The Postgres memory manager "palloc" must be used for all space
 * allocation.  Do not use "malloc".
 */

    result = (CIRCLE *) palloc(sizeof(CIRCLE));
    result->center.x = atof(coord[0]);
    result->center.y = atof(coord[1]);
    result->radius = atof(coord[2]);

    sprintf(buf2, "circle_in: read (%f, %f, %f)0, result->center.x,
    result->center.y,result->radius);
    write(2, buf2, strlen(buf2));

    return(result);
}

char *
circle_out(circle)
CIRCLE    *circle;
{
    char    *result;
```

```
    if (circle == NULL)
        return(NULL);

    result = (char *) palloc(60);
    (void) sprintf(result, "(%g,%g,%g)",
                circle->center.x, circle->center.y, circle->radius);
    return(result);
}

char
circle_area_equal(circle1, circle2)
    CIRCLE    *circle1;
    CIRCLE    *circle2;
{
    return(ABS(circle1->radius - circle2->radius) < 0.0001);
}
```